Examination of the module MPRI 2-30
Cryptographic protocols: formal and computational proofs

(Solution)
November 21, 2018

2 CryptoVerif

2.1 Exercise 1

(1) input and output of $f_{pk}$: $l_0$ (input because of the length of $r$; the output has the same length as the input because $f_{pk}$ is a permutation).

input of $H$: $l_0$; output of $H$: $l_1$ (same length as $m$ because of the $\oplus$)

inputs and output of $\oplus$: $l_1$ (because of $m$)

input of $G$: $l_0 + l_1$; output of $G$: $l_2$.

(2) Let $a \parallel b \parallel c$ be the ciphertext (length of $a = l_0$, length of $b = l_1$, length of $c = l_2$). $r = f_{sk}^{-1}(a)$, $m = H(r) \oplus b$. One can check that $c = G(r \parallel m)$.

(3) type $Dr$ has size $l_0$, type $Dm$ has size $l_1$, type $DGin$ has size $l_0 + l_1$, type $DGout$ has size $l_2$.

let hashoracleG(hkG) = !iG <= qG in(chG1, x:DGin); out(chG2, G(hkG,x)).

let hashoracleH(hkH) = !iH <= qH in(chH1, x:Dr); out(chH2, H(hkH,x)).

let processT(hkH: hashkey, hkG: hashkey, pk: pkey, b1: bool) =
in(chT1, (m1:Dm, m2:Dm));
let menc = if b1 then m1 else m2 in
new r: Dr;
let aT = $f(pk, r)$ in
let bT = xor(H(hkH,r), menc) in
let cT = G(hkG,concat(r,menc)) in
let ciphertext: bitstring = (aT,bT,cT) in
out(chT2, ciphertext).

process
in(start, ());
new hkH: hashkey;
new hkG: hashkey;
new b0: bool;
new rk: keyseed;
let pk = pkgen(rk) in
let sk = skgen(rk) in
out(c1, pk);
(processT(hkH, hkG, pk, b0) \| hashoracleH(hkH) \| hashoracleG(hkG))
(4) We need to add a decryption oracle:

```plaintext
def processD(hkH: hashkey, hkG: hashkey, sk: skey) =
  ! iD <= qD
  in(chD1, ciphertext’ : bitstring);
  (* The attacker cannot call the decryption oracle on the test ciphertext *)
  find such that defined(ciphertext) && ciphertext’ = ciphertext then yield else
  let (a:Dr, b:DM, c:DGout) = ciphertext’ in
  let r = invf(sk, a) in
  let m = xor(H(hkH, r), b) in
  if c = G(hkG, concat(r, m)) then
    out(chD2, m).
processD(hkH, hkG, sk)
```

processD(hkH, hkG, sk) is added to final parallel composition.

(5) Random oracle of $H$ and $G$ can be applied directly. The property of $\oplus$ cannot (even after syntactic transformation) because $\oplus$ has no random argument. One-wayness cannot (even after syntactic transformation) because $f^{-1}$ is used.

Applying the random oracle assumption on $H$ replaces $H(r)$ with a fresh random value $r'$, which helps applying the assumption on $\oplus$. Applying the random oracle assumption on $G$ and $H$ also makes comparisons with $r$ appear, which is helpful to apply one-wayness (and allows CryptoVerif to remove the computation of $r = f^{-1}_sk(a)$ in decryption by simplification).

2.2 Exercise 2

(1) A:

Message 1. $A \rightarrow Y : M, A, Y, \{N_a, M, A, Y\}_{K_{as}}$  
$M, N_a$ fresh

Message 4. $Y \rightarrow A : M, \{N_a, K_{ay}\}_{K_{as}}$

B:

Message 1. $X \rightarrow B : M, X, B, C$

Message 2. $B \rightarrow S : M, X, B, C, \{N_b, M, X, B\}_{K_{bs}}$  
$N_b$ fresh

Message 3. $S \rightarrow B : M, C', \{N_b, K_{xb}\}_{K_{bs}}$

Message 4. $B \rightarrow A : M, C$

S:

Message 2. $Y \rightarrow S : M, X, Y, \{N_x, M, X, Y\}_{K_{sx}}, \{N_y, M, X, Y\}_{K_{sy}}$

Message 3. $S \rightarrow Y : M, \{N_x, K_{xy}\}_{K_{sx}}, \{N_y, K_{xy}\}_{K_{sy}}$  
$K_{xy}$ fresh

(2) let processA(Kas: key) =

```plaintext
def processA(Kas: key) =
in(c1, hb: host);
new M : nonce;
new Na : nonce;
let ea1 = enc(concat1(Na, M, A, hb), Kas) in
out(c2, (M, A, hb, ea1));
in(c3, (=M, ea2: bitstring));
let injbot(concat2(=Na, k)) = dec(ea2, Kas) in
if hb = B then
```
(let keyA:key = k)
else
(out(cAK, k)).

let processB(Kbs: key) =
in(c4, (M: nonce, ha: host, =B, ea:bitstring));
new Nb : nonce;
let e2 = enc(concat1(Nb, M, ha, B), Kbs) in
out(c5, (M, ha, B, ea, e2));
in(c6, (=M, ea2: bitstring, eb2: bitstring));
let injbot(concat2(=Nb, k)) = dec(eb2, Kbs) in
if ha = A then
(let keyB:key = k in out(c7, (M, ea2))
else
(out(c7, (M, ea2, k))).

let processK(Kas: key, Kbs: key) =
in(c8, (Khost: host, Kkey: key));
if Khost = A then insert keys(A,Kas) else
if Khost = B then insert keys(B,Kbs) else
insert keys(Khost,Kkey).

let processS =
in(c9, (M: nonce, ha: host, hb: host, ea1: bitstring, eb1: bitstring));
get keys(=ha, kas) in
get keys(=hb, kbs) in
let injbot(concat1(Na, =M, =ha, =hb)) = dec(ea1, kas) in
let injbot(concat1(Nb, =M, =ha, =hb)) = dec(eb1, kbs) in
new k: key;
let e1 = enc(concat2(Na, k), kas) in
let e2 = enc(concat2(Nb, k), kbs) in
out(c10, (M, e1, e2)).

process
in(start, ());
new Kas: key;
new Kbs: key;
out(c13, ());
((! iA <= N processA(Kas)) |
(!! iB <= N processB(Kbs)) |
(!! iS <= N processS) |
(!! iK <= N2 processK(Kas, Kbs)))

(3) The keys established in various sessions of the protocol are indistinguishable from fresh independent random keys.

(4) query secret keyB.
query secret keyA.

Note that we store the key in a specific variable when it is exchanged between $A$ and $B$, because we cannot prove secrecy for keys that are not for sessions between $A$ and $B$. We leak those keys. (That what is usually done to make sure that there cannot be confusions between keys of different sessions.)
The key is secret. Only $S$ can decrypt the ciphertexts from $A$ and $B$, and only $A$, resp. $B$ can decrypt the server reply. $A$’s request is tied to nonce $N_a$, so that when $A$ sees the server reply, it can be sure to which request it corresponds, and in particular that it is a request to talk to $B$. Similarly, $B$’s request is tied to nonce $N_b$. The nonce $M$ ties the two requests of $A$ and $B$ together.